Escola de Matemática Aplicada

Aula Prática 5 - ALN - Fatoração QR

Aluno: *Gustavo Murilo Cavalcante Carvalho*

E-mail: *gustavomurilo012@gmail.com*

Programa: *Bacharelado em Matemática Aplicada*

Disciplina: *Álgebra Linear Numérica*

Professor: *Antonio Carlos Saraiva Branco*

Data: *16 de junho de 2024*

Sumário

Problema 1

O método de Gram Schmidt para ortogonalização de vetores é um método iterativo que, dado um conjunto de vetores linearmente independentes, gera um conjunto de vetores ortogonais. De modo que cada vetor de Q é obtido subtraindo as projeções do vetor atual sobre os vetores ortogonais anteriores.

Implementação

qr_GS.m

```
% Entradas:
%   A - matriz (m x n)
% Saídas:
%   Q = matriz (m x n) ortogonal
%   R = matriz (n x n) triangular superior
function [Q,R] = qr_GS(A)
  [m, n] = size(A);

  % Inicializa matrizes
  Q = zeros(m,n);
  R = zeros(n);

  for j = 1 : n
    v = A(:,j); % j-ésima coluna de A

    % Obtém, por Gram-Schmidt, v o j-ésimo vetor de uma base ortogonal
    for i = 1 : j-1
      R(i,j) = dot(Q(:,i), A(:,j));
      v = v - R(i,j) * Q(:,i);
    end

    R(j,j) = norm(v,2);
    Q(:,j) = v / R(j,j); % j-ésimo vetor de uma base ortonormal
  end
end
```

Testes

A seguir estão algumas matrizes selecionadas para testar as funções implementadas neste trabalho.

$$A = \begin{bmatrix} 1 & 2 & 2 \\ -1 & 1 & 2 \\ -1 & 0 & 1 \\ 1 & 1 & 2 \end{bmatrix} \quad B = \begin{bmatrix} 16 & 2 & 3 & 13 \\ 5 & 11 & 10 & 8 \\ 9 & 7 & 6 & 12 \\ 4 & 14 & 15 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 2 & 1 & 2 & 4 \\ 3 & 1 & 2 & 2 \end{bmatrix}$$

A é uma matriz ideal para ortogonalização (tanto que é um exemplo dado pelo Poole), pois contém vetores para uma base de um subespaço vetorial.

B é uma matriz mágica de ordem par, portanto é muito má condicionada (ela é quase singular), o que é interessante para os testes.

C é uma matriz retangular com mais colunas do que linhas, o que não é o caso esperado de acordo com a apresentação da fatoração QR (a que é feita no Poole), contudo é um cenário de prova para a implementação e interpretação do método.

Testando para o cenário ideal (a matriz A), temos:

```
>> [QCa, RCa] = qr_GS(A);

QCa =
    0.5    0.67082   -0.40825
   -0.5    0.67082          0
   -0.5    0.22361    0.40825
    0.5    0.22361     0.8165

RCa =
    2        1       0.5
    0   2.2361    3.3541
    0        0    1.2247
```

Considerando $A = QR$, para verificar a ortogonalidade de Q, calculamos Q^TQ e para verificar a acurácua decomposição QR, calculamos $QR - A$.

```
>> QCa'*QCa
    1          0          0
    0          1   -2.7e-17
    0   -2.7e-17          1

>> QCa*RCa - A
    0    0    0
    0    0    0
    0    0    0
    0    0    0
```

Pode ser visto que a decomposição QR obtida foi muito boa. Q não é por muito pouco (o erro é irrelevante, tem grandeza 10^{-17}) a indentidade, e a multiplicação de Q e R resulta em A.

Para as outras matrizes (B e C), temos:

Ambas são boas fatorações, afinal a multiplicação das matrizes resulta na matriz original, ou algo muito próximo disso. Vale ressaltar que a fatoração funciona para C (uma matriz com mais colunas do que linhas), mesmo que esvaziada do sentido.

```
>> QCb*RCb - B
    0    0    0    0
    0    0    0    0
    0    0    0    0
    0    0    0    0

>> QCc*RCc - C
   -2.2e-16    0    0    0
          0    0    0    0
```

Testando a ortogonalidade de B, vemos que o resultado não é o melhor, muitas entradas são muito próximas de zero, outras nem tanto (na ordem de 10^{-1}). Isso é uma consequência do mal condicionamento de B.

```
>> QCb'*QCb
          1    -2.7e-17     4.9e-16     0.55125
   -2.7e-17           1    -5.5e-16    -0.25841
    4.9e-16    -5.5e-16           1     -0.7925
    0.55125    -0.25841     -0.7925           1
```

Para C, temos que avaliar algo diferente, afinal a matriz Q associada a ela não pode ser ortogonal, ela não é LI. Note que há um bloco que é a indentidade, o que acontece devido ao fato de que os dois primeiros vetores são LI, se fossem outros, o bloco da identidadde estaria em outra posição (ao menos é .

```
>> QCc'*QCc
          1     1.4e-15     0.00377     0.00377
    1.4e-15           1    -0.99999    -0.99999
    0.00377    -0.99999           1           1
    0.00377    -0.99999           1           1

>> QCc(1:2, 1:2)'*QCc(1:2, 1:2)
          1     1.4e-15
    1.4e-15           1
```

Problema 2

A versão modificado tem como objetivo trazer uma melhor establidade numérica ao método de Gram-Schmidt. A ideia é que, ao invés de calcular a projeção de um vetor sobre os outros já ortogonalizados, calculamos a projeção do vetor mais atualizados sobre os já ortogonalizados. Isso evita a acumulação de erros numéricos.

Implementação

qr_GSM.m

```
% Entradas:
%   A - matriz (m x n)
% Saídas:
%   Q = matriz (m x n) ortogonal
%   R = matriz (n x n) triangular superior
function [Q,R] = qr_GSM(A)
  [m, n] = size(A);

  % Inicializa matrizes
  Q = zeros(m,n);
  R = zeros(n);

  for j = 1 : n
    v = A(:,j); % j-ésima coluna de A

    % Obtém, por Gram-Schmidt, v o j-ésimo vetor de uma base ortogonal
    for i = 1 : j-1
      R(i,j) = dot(Q(:,i), v); % Passa a usar o vetor mais atualizado
      v = v - R(i,j) * Q(:,i);
    end

    R(j,j) = norm(v,2);
    Q(:,j) = v / R(j,j); % j-ésimo vetor de uma base ortonormal
  end
end
```

Testes

Para a matriz A, essa modificação obteve os mesmos resultaos que a anterior. Portanto, não há necessidade expor os resultados.

Para a matriz B, uma matriz má condicionada, a modificação alterou minimamente uma entrada da matriz R. Quanto à matriz Q, notamos que a ultima colunas dessa nova Q é diferente da anterior.

```
>> QCb(:,4)' % Quarta coluna da matriz Q obtida com qr_GSC
    0.32233    0.40291    0.64466   -0.56408

>> [QMb, RMb] = qr_GSM(B);

>> QMb(:,4)'
    0.94679    0.063119    0.25248   -0.18936
```

Associada a essa mudança, vemos uma melhor ortogonalidade da matriz Q utilizando o método de Gram-Schmidt modificado. Contudo, devido à dimensão da matriz esse ganho é mínimo.

Considerando matrizes mágicas de ordem par (caracterizadas por serem muito mal condicionadas), fica claro que esse algoritmo produz uma matriz Q muito mais próxima da ortogonalidade.

```
>> [Q ~] = qr_GS(magic(20));
>> [QM ~] = qr_GSM(magic(20));
>> norm(Q'*Q - eye(20))
    16.989
>> norm(QM'*QM - eye(20))
    0.99972

>> [Q ~] = qr_GS(magic(100));
>> [QM ~] = qr_GSM(magic(100));
>> norm(Q'*Q - eye(100))
    97
>> norm(QM'*QM - eye(100))
    0.99995
```

Resta testar a matriz C. Agora, com essa modificação, deixamos de obter uma fatoração eficaz. Ambas as matrizes possus entradas NaN (não faz sentido verificar ortogonalidade ou acurácia). Isso se deve ao fato de C possuir colunas LD (linearmente dependentes). Logo essa modificação do método restringe a função a matrizes LI (linearmente independentes).

```
>> [QMc, RMc] = qr_GSM(C)
QMc =
     0.5547    0.83205    NaN    NaN
    0.83205    -0.5547    NaN    NaN

RMc
    3.6056     1.3868    2.7735    3.8829
         0    0.27735    0.5547    2.2188
         0          0         0       NaN
         0          0         0       NaN
```

Problema 3

Implementação

qr_GSP.m

O objetivo do pivoteamento parcial, assim como quando usado na fatoração LU, é melhorar a estabilidade numérica. Portanto, como implementada abaixo, essa função é uma versão melhorada de `qr_GSM`.

```
% Entradas:
%   A - matriz (m x n)
% Saídas:
%   Q = matriz (m x n) ortogonal
%   R = matriz (n x n) triangular superior
function [Q, R, P] = qr_GSP(A)
  [m, n] = size(A);
  Q = zeros(m, n);
  R = zeros(n, n);
  P = eye(n); % Matriz de permutação

  for j = 1:n
    % Encontra o índice da coluna com a maior norma
    [~, indice_max] = max(vecnorm(A(:, j:n)));
    indice_max = indice_max + j - 1;

    % Trocar colunas de A
    if indice_max ~= j
      A(:, [j, indice_max]) = A(:, [indice_max, j]);
      P(:, [j, indice_max]) = P(:, [indice_max, j]);
    end

    v = A(:,j); % j-ésima coluna de A

    % Obtém, por Gram-Schmidt, v o j-ésimo vetor de uma base ortogonal
    for i = 1 : j-1
      R(i,j) = dot(Q(:,i), v); % Usa o vetor atualizado
      v = v - R(i,j) * Q(:,i);
    end

    R(j,j) = norm(v,2);
    Q(:,j) = v / R(j,j); % j-ésimo vetor de uma base ortonormal
  end

end
```

Testes

Como pode ser visto os resultados para todas as matrizes são bem diferentes das questões anteriors, o que significa que foram feitas trocas de colunas, o pivoteamento entrou em ação.

```
>> [QPa, RPa, Pa] = qr_GSP(A)
QPa =
      0.5547     0.74125    -0.09759
      0.5547    -0.22237    -0.58554
     0.27735      -0.593    -0.19518
      0.5547    -0.22237     0.78072

RPa =
    3.6056     2.2188    0.27735
         0     1.0377     1.3342
         0          0     1.4639
```

```
>> [QPb, RPb, Pb] = qr_GSP(B)
QPb =
    0.82295   -0.37754    -0.36086   -0.41646
    0.25717    0.54622      0.4307          0
    0.46291    0.57836    0.034922   -0.36441
    0.20574   -0.47393     0.82648   -0.83293

RPb =
    19.442     18.516      10.904      10.595
         0     5.9281     0.69081      2.6669
         0          0      15.831      15.831
         0          0           0    2.1e-15
```

```
>> [QPc, RPc, Pc] = qr_GSP(C)
QPc =
    0.89443   -0.44721   -0.24254     NaN
    0.44721    0.89443    0.97014     NaN

RPc =
    4.4721     3.1305    2.6833      1.3416
         0     1.7889    0.89443    0.44721
         0          0    2.2e-16    1.1e-16
         0          0          0          0
```

Note que, assim como no problema anterior, aqui C apresenta valores NaN. Por isso, deixamos de fazer, nesse exemplo, os testes para essa matriz.

Agora resta verificar a acurárcia da fatoração e a ortogonalidade das matrizes resultantes.

```
>> norm((QPa * RPa) * Pa - A)
   2.2e-16

>> norm((QPb * RPb) * Pb - B)
   8.9e-16
```

```
>> norm((QPa' * QPa) - eye(3))
    3.7e-16

>> norm((QPb' * QPb) - eye(4))
    0.9413
```

Por fim, repetirei o teste para matrizes má condicionas feito para o problema anterior. No teste em questão, vemos que que o desempenho de `qr_GSP` é inferior ao de `qr_GSM`. Pois a Q_P está mais longe de ser ortogonal do qeu Q_M.

```
>> [QM ~] = qr_GSM(magic(20));
>> norm(QM'*QM - eye(20))
    0.99972
>> [QP, ~, ~] = qr_GSP(magic(20));
norm(QP'*QP - eye(20))
ans =
    0.99993

>> [QM ~] = qr_GSM(magic(100));
>> norm(QM'*QM - eye(100))
    0.99995
>> [QP, ~, ~] = qr_GSP(magic(100));
>> norm(QP'*QP - eye(100))
    1
```

Problema 4

Implementação

qr_House_1.m

Esta é a implementação padrão do método de Householder para fatoração QR. Tranforsma a matriz $A \in \mathbb{M}^{m \times n}$ em uma matriz triangular superior $R \in \mathbb{M}^{m \times n}$ e salvamos os refletores usados para obter Q posteriormente. Essa primeira versão de implementação só abrange os casos onde $m \geq n$.

```
% Entradas:
%   A - matriz de entrada (m x n)
% Saídas:
%   U - matriz (m x m) contendo os vetores normais
%   R - matriz (m x n) triangular superior
function [U, R] = qr_House_1(A)
  [m, n] = size(A);

  % Inicializar a matriz U com zeros
  U = zeros(m, m);

  for i = 1 : n
    % Extrair a coluna atual a partir da i-ésima linha até o final
    x = A(i:m, i);

    % Obtém o vetor normal ao hiperplano de reflexão
    if x(1) > 0
      x(1) = x(1) + norm(x, 2);
    else
      x(1) = x(1) - norm(x, 2);
    end

    u = x / norm(x, 2); % Normaliza o vetor
    U(i:m, i) = u;  % Armazena o vetor em U

    % Aplica a transformação de Householder à submatriz de A
    A(i:m, i:n) = A(i:m, i:n) - 2*u*(u'*A(i:m, i:n));
  end

  R = triu(A); % Os valores abaixo da diagonal seriam proximos de 0, trunco-
os
end
```

qr_House_2.m

Seja $A \in M^{m \times n}$. Para quaisquer $m, n \in \mathbb{N}$ a função `qr_House_2(A)` é capaz de realizar parte da fatoração, o que é feito por completo ao construi Q a partir de U.

Se $m > n$: a função obtém o mesmo resultado que qr_GS.

Se $m = n$: temos $Q \in \mathbb{M}^{m \times m-1}$ e $R \in \mathbb{M}^{m-1 \times n}$.

Se $m < n$: temos $Q \in \mathbb{M}^{m \times m}$ e $R \in \mathbb{M}^{m \times n}$, o resultado geral de qr_House_1, mas agora funcionando para esse caso.

```
% Entradas:
%   A - matriz (m x n)
% Saídas:
%   U - matriz (m x k) contendo os vetores normais
%   R - matriz (k x n) triangular superior
function [U,R] = qr_House_2(A)
  [m, n] = size(A);

  % Determina a dimensão correta
  if m == n
    k = m - 1;
  else
    k = min(m,n); % Essa alteração abrange o caso onde m < n
  end

  % Inicializa matrizes
  U = zeros(m, k);

  for i = 1 : k
    x = A(i:m, i);

    % Obtém o vetor normal ao hiperplano de reflexão
    if x(1) > 0
      x(1) = x(1) + norm(x, 2);
    else
      x(1) = x(1) - norm(x, 2);
    end

    u = x / norm(x,2); % Normaliza o vetor
    U(i:m, i) = u;     % Armazena o vetor em U

    % Aplica a transformação de Householder à submatriz de A
    A(i:m, i:n) = A(i:m, i:n) - 2*u*(u'*A(i:m, i:n));
  end

  R = triu(A(1:k, 1:n)); % Para que seja compatível com a matriz R de qr_GS
end
```

constroi_Q.m

```
% Entradas:
%   U - matriz (m x n) com vetores de Householder
% Saídas:
%   Q - matriz (m x n) ortogonal
%   versao - 1: Q é m x m, 2: Q é m x n
function [Q] = constroi_Q(U, versao)
  % Obtém dimensões de U e inicializa Q
  [m,n] = size(U);

  Q = eye(m, m);

  for i = 1 : n
    u = U(:,i);

    % Aplica a tranformação de Householder pela direita Q*(H - u*u')
    Q = Q - 2 * Q * (u * u');
  end

  % Ajusta a dimensão de Q para ser compatível com R
  if versao == 2
    Q = Q(:,1:n);
  end
end
```

Testes

Testes - $A(4 \times 3)$

Como pode ser visto no exemplo a seguir, a segunda versão retorna um "recorte" do que seria retornado pela primeira versão. Nesse cenário, fica bem claro que isso é, na verdade, uma seleção da informação relevante, afinal foram cortadas as colunas ou linhas de zero.

```
>> [UHa, RHa] = qr_House_1(A)
UHa =
    0.86603          0          0    0
   -0.28868    0.97325          0    0
   -0.28868    0.22975    0.80756    0
    0.28868    1.0e-16    0.58979    0

RHa =
   -2        -1   -0.5
    0   -2.2361   -3.3541
    0         0   -1.2247
    0         0    0
```

```
>> [UKa, RKa] = qr_House_2(A)
UKa =
    0.86603          0          0
   -0.28868    0.97325          0
   -0.28868    0.22975    0.80756
    0.28868    1.0e-16    0.58979

RKa =
   -2         -1    -0.5
    0    -2.2361    -3.3541
    0          0    -1.2247
```

Construindo as matrizes Q correspondentes, vemos que ambas são muito próximas de serem ortogonais, sem diferença significativa.

```
>> QHa = constroi_Q(UHa, 1) % Versão 1
QHa =
   -0.5   -0.67082    0.40825   -0.36515
    0.5   -0.67082    4.4e-16    0.54772
    0.5   -0.22361   -0.40825    -0.7303
   -0.5   -0.22361    -0.8165    0.18257

>> norm(QHa' * QHa - eye(4)) % Teste de ortogonalidade
    1.7e-15

>> QKa = constroi_Q(UKa, 2)  % Versão 2
QKa =
   -0.5   -0.67082    0.40825
    0.5   -0.67082    4.4e-16
    0.5   -0.22361   -0.40825
   -0.5   -0.22361    -0.8165

>> norm(QKa' * QKa - eye(3)) % Teste de ortogonalidade
    1.6e-15
```

Ao multiplicar as matrizes, o resultado indifere da função. Obtemos umas matriz quase igual a A, porém com $9.9e-16$ ao invés de 0 em uma das entradas, um erro insignificante que pode ser desconsiderado.

Testes - $B(4 \times 4)$

Para B a fatoração funciona e as duas versões tem comportamento similar ao apresentado no exemplo anterior.

Quanto à ortogonalidade e à acurárcia, vemos que essa decomposição é precisa.

```
>> norm(QHb * RHb - B)         % Versão 1
    1.9e-14
>> norm(QHb' * QHb - eye(4))  % Versão 1
    7.5e-16
>> norm(QKb * RKb - B)         % Versão 2
   1.8e-14
>> norm(QKb' * QKb - eye(3))  % Versão 2
   6.8e-16
```

Para matrizes má condicionadas de ordem eleveda, ambas as funções apresentam o mesmo resultado, as matrizes da fatoração vão igualmente bem em todos os testes. Fiz o experimento com uma matriz mágica de ordem 100.

Testes - $C(2 \times 4)$

A primeira versão de qr_House não é capaz de lidar com matrizes com esses dimensões, onde $n > m$. Assim, só apresentarei os testes com qr_House_2.

```
>> [UKc, RKc] = qr_House_2(C)
UKc =
    0.88167    0
    0.47186   -1

RKc =
    -3.6056    -1.3868   -2.7735   -3.8829
          0    0.27735    0.5547    2.2188

>> QKc = constroi_Q(UKc, 2)
QKc =
    -0.5547    0.83205
   -0.83205    -0.5547
```

Perceba que esta também é uma fatoração bem sucedida. Temos que $C = QR$ e Q é praticamente ortogonal.

```
>> QKc * RKc
    2    1    2    4
    3    1    2    2

>> norm(QKc * RKc - C)
    7.8e-15

>> norm(QKc' * QKc - eye(2))
    1.1e-15
```

Problema 5

Implementação

espectro.m

Vale comentar que optei pela função qr_GSM para fazer a fatoração QR por ser mais estável que a qr_GS e mais simples (não é preciso construir Q a partir de U) do que as fatorações que utilizam os refletores de Householder.

```
% Entradas:
%   A - matriz (n x n)
% Saídas:
%   S = vetor (n x 1) ortogonal
function [S] = espectro(A, tol)
  % Definição de variáveis
  erro = tol + 1;
  S = diag(A);

  while tol <= erro
    % Processo iterativo
    [Q, R] = qr_GSM(A);
    A = R * Q;

    % Verificação de convergência
    novo_S = diag(A);
    erro = norm(S - novo_S,'inf');

    S = novo_S; % Atualiza o resultado
  end
end
```

Testes

Para os testes, gero matrizes com números inteiros uniformemente distribuídos entre 1 e 9. A matriz é então multiplicada por sua transposta para que seja simétrica e, portanto, tenha autovalres reais. Então comparamos os autovalores obtidos pela função criada com os autovalores obtidos pela função `eig` do MATLAB.

Todos os testes a seguir foram feitos com uma tolerância de $10^{\{-12\}}$, e mesmo assim os resultados foram obtidos muito rapidamente.

```
>> M = randi(9,5,5);

>> M = M' * M;

>> flip(eig(M))
    606.58    61.75    29.118    4.9587    0.58848

>> S = espectro(M, 1e-12)
    606.58    61.75    29.118    4.9587    0.58848
```

Uma ideia interessante para escalar os testes para matrizes maiores, é verificar a norma entre a diferença do resultado das funções, ao invés de comparar os vetores diretamente.

```
>> N = randi(9,10,10);
>> N = N' * N;
>> S = espectro(N, 1e-12);
>> S - flip(eig(N))
    9.0949e-13
   -8.5265e-14
    1.4211e-14
   -1.4211e-14
    5.6843e-14
    3.5527e-14
   -1.3603e-11
    1.3443e-11
   -3.1974e-14
    1.9054e-13

>> norm(espectro(N,1e-12) - flip(eig(N)))
    1.9148e-11
```

Verificando para uma matriz de ordem 100, temos:

```
>> O = randi(9,200,200)
>> O = O' * O;
>> norm(espectro(O,1e-12) - flip(eig(O)))
    7.5866e-10
```

Esse resultado foi bem precismo, mas a função já demorou bem mais para convergir (aproximadamente 1 minuto).

Para matrizes maiores, a nossa função passa a ser muito lenta o que torna o seu uso inviável. Por exemplo, para uma matriz de ordem 300 o algoritmo demorou quase 6 minutos para convergir.

Conclusão

Essa sessão tem como objetivo, partindo dos dados dos testes da páginas anteriores, definir quais implementações são as mais desejáveis para realizar a fatoração QR.

Para os exemplos usados, tivemos melhoria significativa entre `qr_GS` e `qr_GSM`. Acrescentar mais uma etapa, o pivoteamento parcial não se mostrou muito útil, pode ser que eu não tenha testado com um exemplo conveniente, mas não me parece o caso.

Quanto à acurácia e ortogonalidade, não percebi nenhuma diferença significativa entre usar uma função baseada no método de Gram-Schmidt ou Householder.

Para lidar com matrizes má condicionadas, as funções `qr_GSM`, `qr_GSP`, `qr_House_1` e `qr_House_2` estão equiparadas, a diferença entre elas não é tão relevante, pois quando grande, a diferença fica na escala de 10^2 ou 10^3.

Por fim, ressalto as funções preferíveis em cada caso:

Se $A \in \mathbb{M}^{m\times n}$, com $m \geq n$:

É recomendado o uso de `qr_GSM`, afinal é o caso mais simples dentre os que têm o seu nível de precisão.

Se $A \in \mathbb{M}^{m\times n}$, com $m < n$:

Prefira o uso de `qr_House_2`, pois esse caso, diferente de vários outros, é capaz de retorna uma fatoração funcional. Para o outro caso ($m \geq n$), essa versão faz com que o resultado seja o mesmo que o do método de GS. Para o cenário em questão, essa fatoração se parece com a fatoração reotrnada por `qr_House_1` para matrizes nas quais $m \geq n$